Plan Execution for Autonomous Spacecraft [*]

Barney Pell [†] Erann Gat [§] Ron Keesing [†] Nicola Muscettola [‡] Ben Smith [§]

Abstract

The New Millennium Remote Agent (NMRA) will be the first AI system to control an actual spacecraft. The spacecraft domain raises a number of challenges for planning and execution, ranging from extended agency and long-term planning to dynamic recoveries and robust concurrent execution, all in the presence of tight real-time deadlines, changing goals, scarce resource constraints, and a wide variety of possible failures. We believe NMRA is the first system to integrate closed-loop planning and execution of concurrent temporal plans, and the first autonomous system that will be able to achieve a sustained multi-stage multi-year mission without communication or guidance from earth.

1 Introduction

We are developing the first AI system to control an actual spacecraft. The mission, Deep Space One (DS-1) is the first mission in NASA's New Millennium Program (NMP), an aggressive series of technology demonstrations intended to push Space Exploration into the twenty-first century. DS-1 will launch in mid-1998 and will navigate and fly by asteroids and comets, taking pictures and sending back information to scientists back on Earth. One key technology to be demonstrated is spacecraft autonomy, including onboard planning and plan execututon. The spacecraft will spend long periods of time without the possibility of communication with ground operations staff, and will in fact plan when it needs to communicate back to Earth. It must maintain its safety and achieve high-level goals, when possible, even in the presence of hardware faults and other unexpected events.

This paper describes our approach to plan execution in the context of spacecraft autonomy. Our approach is being implemented as part of the New Millennium Remote Agent (NMRA) architecture (Pell *et al.* 1996). The paper is organized as follows. Section 2 discusses the spacecraft domain and requirements which influence our design. Section 3 describes our approach to planning, execution, and robustness, and illustrates the top-level loop of our system. Section 4 addresses the issues involved in generating plans to support robust execution, and Section 5 shows how such plans are executed. We then consider our work in the context of related work and conclude.

2 Domain and Requirements

The autonomous spacecraft domain presents a number of challenges for planning and plan execution. Many devices and systems must be controlled, leading to multiple threads of complex activity. These concurrent processes must be coordinated to control for negative interactions, such as the vibrations of the thruster system violating the stability requirements of the camera. In addition, activities may have precise real-time constraints, such as taking a picture of an asteroid during a narrow window of observability.

The planner and plan execution system must reason about and interact with external agents and processes, such as the on-board navigation system and the attitude controller. These external agents can provide some information at plan time, and achieve tasks and provide more information at runtime, but are never fully controllable or predictable. For example, the attitude controller can provide estimates of turn durations at plan time. However,

[*]This paper appears in the working notes of the AAAI Fall Symposium on Plan Execution

[‡]Recom Technologies, NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.

[†]Caelum Research, NASA Ames Research Center, MS 269/2, Moffett Field, CA 94035.

[§]Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Pasadena, CA 91109.

the completion of turns during execution is not controllable and can only be observed. Plans must express compatibilities among activities, and the plan execution system must synchronize these activities at run-time.

Virtually all resources on spacecraft are limited and carefully budgeted, and the system must ensure that they are allocated effectively to goal-achievement. Some resources, like solar panel-generated power, are constant but limited. Others, such as total propellant, are finite and must be budgeted across the entire mission. The planner reasons about resource usage in generating plans, but because of run-time uncertainty the resource contraints must also be enforced as part of plan execution.

In addition, planning and the information necessary to generate plans can also be considered limited resources. Thus plan execution must be robust in the face of a wide variety of hardware faults and delays.

The spacecraft domain requires tracking of complex goals through time, such as the achievement of a certain amount of thrusting over the course of several months. Thus the execution system must communicate the outcomes of commanded activities that affect these goals to the planning system, and the planner must use this information to update current goals and influence future planning.

3 Approach

Our approach clearly separates an extensive, deliberative planning phase from the reactive execution phase. The characteristics of the resulting architecture stem from the challenge of solving the complex technical problems posed by the domain within the compressed implementation timeline imposed by the mission.

Our architecture follows the approach of infrequently generating plans over relatively extended periods of time.

Because on-board resources are severely limited the planner needs to generate courses of action that achieve high quality execution and cover extended periods of time. The overall goal of "mission success" depends on the achievement of several individual goals imposed on the mission from several independent sources (e.g., take pictures with a given frequency to perform optical navigation, communicate to ground at given times). However these goals cannot usually be achieved with the level of satisfaction required by each source (e.g., required duration of a communication activity). The planner always has to trade off the level of goal satisfaction with respect to the long term "mission success" and within the resource limitations. Degradation or even rejection of individual goals is a likely outcome of planning. The above trade-offs can be reasonably evaluated only by considering extended periods of plan execution.

The length of the horizon covered by each plan needs to be carefully dimensioned to take into account the effects of execution uncertainty. To achieve some planning goals (e.g., accumulation of thrust) the planner relies on predictions of the behavior of the spacecraft (e.g., nominal schedules of engine activation provided by the on-board navigator). However the need to achieve other, independent goals is not included in the assumptions under which the prediction is generated. To be completely accurate the prediction should take into account the detailed activities (e.g., when the engine will be actually switched on or off) and these are only known in the final plan . In our approach the predictive models are updated at periodic times which correspond with the end of each scheduling horizon.

We choose to plan at infrequent intervals because of the limited on-board processing capabilities of the spacecraft. The planner must share the CPU with other critical computation tasks such as the execution engine, the real-time control loops and the fault detection, isolation and recovery system. While the planner generates a plan, the spacecraft must continue to operate. More importantly, the plan often contains critical tasks whose execution cannot be interrupted in order to install newly generated plans. In our approach the generation of a plan is explicitly represented in the plan as a task. When the executive reaches this task in the current planning horizon, it asks the planner to generate a plan for the next scheduling horizon while it continues to execute the activities remaining in the current plan. In nominal condition the planning activity is guaranteed to generate a plan within its allocated duration. When the executive reaches the end of the current horizon the plan for the next horizon will be ready and the executive will be able to install it and to seamlessly continue execution.

Ideally, we would like to have the planner represent the spacecraft at the same level of detail as the executive. This approach is taken by (Bresina *et al.* 1996), and by (Levinson 1994). The approach, when feasible, has a number of benefits. First, it enables the planner to simulate the detailed functioning of the executive under various conditions of uncertainty, and to produce a plan which has contingencies (branches) providing quick responses for important execution outcomes. Second, it enables one language rather than two for expressing action knowledge, which simplifies knowledge engineering and helps maintain consistency of interfaces. Third, it enables the planner to monitor execution in progress and project the likely course of actions, and then provide plan refinements which can be patched directly into the currently executing plan.

Unfortunately, in our domain this single representation approach is not really practical because the complexity of interactions at the detailed level of execution would make planning at this level combinatorially intractable.

To simplify the job of the planner, we have found it necessary to make the planner operate on a more abstract model of the domain. Examples of abstractions we use include:

- hiding details of subsystem interactions controlled by the executive
- merging a set of detailed component states into abstract states
- not modeling certain subsystems
- using conservative resource and timing estimates

Some examples are these abstractions are provided in Section 6.

These techniques successfully simplify the combinatorial challenges to the point where we can actually fly a planner on-board the spacecraft and have it solve real problems. However, this does create an abstraction boundary between the action models used by the planner and the executive. This language boundary has several consequences which impact our design. One important consequence is that the planner can no longer model or predict intermediate execution states. Since the executive is managing multiple concurrent activities at a level of detail below the planner's visibility, this makes it difficult to provide a well-defined initial state as input to the infrequent planning process. If the executive performed all kind of activities while planning was occuring, it is quite possible that the plan returned is not applicable as the initial conditions are no longer valid at the time the plan is to be executed.

We address the problem of generating initial states for the next planning round differently depending on whether the currently executing plan is succeeding or has failed.

If the currently executing plan is proceeding as planned, it will include an activity to plan for the next horizon. At this point, we can send to the planner the current plan in its entirety, with annotations for the decisions that were made so far in executing it. The current plan serves as its own prediction of the future at the level of abstraction required by the planner. Thus, all the planner has to do is extend the plan to address the goals of the next planning horizon and return the result to the executive. This requires the executive be able to merge in the extended plan within its current representation of the existing plan. The net result is that, from the executive's perspective, executing multiple chained plans is virtually the same as executing one long plan. This has the useful consequence that it enables the executive to engage in activities which span multiple planning horizons (such as a 3-month long engine burn) without interrupting them.

In the event of plan failure, the executive knows how to enter a stable state (called a standby mode) prior to invoking the planner, and it knows how to express that standby mode in the abstract language understood by the planner. It is important to note that establishing standby modes following plan failure is a costly activity, as it causes us to interrupt the ongoing planned activities and lose important opportunities. For example, a plan failure causing us to enter standby mode during the comet encounter would cause loss of all the encounter science, as there is not time to re-plan before the comet is out of sight. Such concerns motivate a strong desire for plan robustness, in which the plans contain enough flexibility, and the executive has the capability, to continue execution of the plan under a wide variety of execution outcomes.

The top-level execution loop can thus be summarized as follows:

1. Begin waiting for signals of plan failure. If this occurs at any time in this sequence, abort the currently executing plan and go to step 2.
2. Begin executing a standby plan to establish a stable state.
3. Invoke planner to generate a new plan, using the current state (including annotations on any currently-executing plan) as the initial state for planning.
4. Continue executing the current plan while waiting for the new one.
5. Upon receipt of the new planner-generated plan: Merge the new planner-generated plan into the current plan.
6. Upon reaching a new planning goal, repeat from step 3.

4 Planning

A principal goal of the NMRA is to enable a new generation of spacecraft that can carry out complete, nominal missions without any communication from ground. This is a great departure from previous and current missions (such as Voyager, Galileo or Cassini) which rely on frequent and extensive communications from ground. In traditional missions, ground operations routinely uplinks detailed command sequences to be executed during subsequent mission phases. Such communications require costly resources such the Deep Space Network, which makes them very expensive. Uplink independence is particularly important for missions that require fast reaction times (as it is the case for autonomous rovers, comet landers and other remote explorers); in this case detailed ground-based commanding is not even feasible due to the long round-trip communication times. Although in case of loss of uplink capabilities previous spacecrafts could carry out a critical sequence of commands stored on board before launch, these sequences were greatly simplified when compared to the uplinked sequences and could only carry out a small fraction of all mission goals.

Mission Manager

NMRA instead is launched with a pre-defined "mission profile". This contains the list of all nominal goals that have to be achieved over the entire duration of the mission. The detailed sequence of commands to acheve such goals, however, is not pre-stored but is generated on board by the planner. A special module of the planner, the Mission Manager, determines the goals that need to be achieved in the next scheduling horizon (typically 2 weeks long), extracts them from the mission profile and combines them with the initial spacecraft state as determined by the executive. The result is the planning problem that once solved yields the detailed execution commands. The mission profile is a significantly more abstract representation of the mission than the complete sequence of detailed commands needed to execute it. The onboard storage requirements for the mission profile, the domain models, and the planning and execution engine is significantly lower than what the storage required by all mission sequences needed in the old approach.

Requirements for Robust Execution

The NMRA must be able to respond to unexpected events during plan execution without having to plan the response. Although it is sometimes necessary to replan, this should not be the only option. Many situations require responses that cannot be made quickly enough if the NMRA has to plan them.

The executive must be able to react to events in such a way that the rest of the plan is still valid. To support this, the plan must be flexible enough to tolerate both the unexpected events and the executive's responses without breaking. This flexibility is achieved in two ways: first, by choosing an appropriate level of abstraction for the activities, and second, by generating plans in which the activities have flexible start and end times.

The abstraction level of the activities in the plan must be chosen carefully. If the activities are at too fine a level of granularity, then the plan will impose too many constraints on the behavior of the executive. However, if the granularity is too course, then there may be interactions among the sub-actions of activities that the planner cannot reason about. In DS1, activities are abstracted to the level where there are no interactions among their sub-activities. This level allows the planner to resolve all of the global interactions without getting into details that would overconstrain the executive.

The other mechanism by which the executive can respond to events without breaking the plan is having activities with flexible start and end times. Plans in DS1 consist of temporal sequences of activities, or tokens. Each activity has an earliest start time, a latest start time, an earliest end time, and latest end time. The planner uses a least commitment approach, constricting the start and end times only when absolutely necessary. Any flexibility remaining at the end of planning is retained in the plan. This flexibility is used by the executive to adjust the start and end times of activities as needed. For example, if the engine does not start on the first try, the executive can try a few more times. To make time for these extra attempts, the end time is moved ahead, but not beyond the latest end time.

Changing the start or end time of an activity may also affect other activities in the plan. For example, if the spacecraft must take science data five minutes after shutting down the engine, then changing the end time of the `engine firing` activity will change the start time of the `take science data` activity. To make the changes, the executive must know about the temporal constraint between the `fire engine` activity and the `take science data` activity. The plan therefore contains all of the temporal constraints among the activities.

Although flexibility in the activity start and end times typically occurs because the times are underconstrained, flexibility can also occur because the duration of an activity is not determined until execution time.

The plan must be able to represent this kind of uncertainty. There are two ways of doing this. One is to use the existing capability for flexible end times. However, the planner assumes that it can further constrain any flexibility in the plan as needed to make room for other activities and constraints. It must remember to preserve the flexibility in the end times–the planner cannot decide to end the warm up early if it needs the time for something else. This enhancement has not yet been implemented for DS1.

A second approach is to fix the end time of the activity to the latest end time, and change the semantics of the activity from `engine warming up` to `engine warming up, and possibly started`. It is then up to the executive to determine whether `warming up` or `started` is the correct state at any time during the activity based on execution time data. This approach requires no change to either the planning or execution engines, and is the approach currently being used by NMRA.

5 Execution

From the point of view of the NMRA executive, a plan is a set of timelines, each of which consists of a linear sequence of tokens. A token consists of a start and end window, a set of pre- and post-conditions, and a token type.

The start and end windows are intervals in absolute time during which that token must start and end. The pre- and post-constraints describe the dependencies of the start and end of the token with respect to the starts and ends of other tokens on other timelines. The token type defines the activity which should be taking place during the time period covered by the token.

There are three different types of pre- and post-constraints, which we will refer to here as *before*, *after*, and *meets* constraints. (The actual nomenclature and representation used is slightly different.) The semantics of these constraints is fairly straightforward. A before/after constraint specifies that the start (or end) of a token must come before/after the start (or end) of another token. The amount of time that may elapse between these two related events is specified as an interval. A meets-constraint specifies that the start (or end) of a token must coincide with the start (or end) of another token. (This is actually represented in the plan as a before- or after-constraint with an allowable elapsed time of zero.)

Issues

Plan execution would be relatively straightforward were it not for the fact that different token types have different execution semantics. In particular, there are different ways of determining whether or not a particular activity has ended or not. Some activities are brought to an end by the physics of the environment or the control system (e.g. turns) while others are brought to an end simply by meeting all its internal plan constraints (e.g. the periods of constant-attitude pointing between turns).

The situation is further complicated by the fact that a naive operationalization of these constraints leads to deadlock. Consider a constant-pointing token A followed by a turn token B. Token A (waiting for the turn) should end whenever token B (the turn) is eligible to start. However, B is constrained by the planner to follow A, and so B is not eligible to start until A ends. Thus, A can never end, and B can never start.

Another issue is that some tokens don't achieve the conditions that they are intended to achieve until some time after they have started. For example, consider a timeline for a device containing a token A of type DEVICE-OFF followed by token B of type DEVICE-ON. The intent here is that the executive should turn the device on at the junction between A and B, but this cannot be done instantaneously. Thus, a token on another timeline constrained to start after B starts on the assumption that the device will be on may fail because the device may not in fact be turned on until some time after B starts.

One possible solution to this problem is to change the planner model so that it generates a plan that includes an intermediate token of type DEVICE-TURNING-ON, but this can significantly increase the size of the planner's search space, and hence the time and resources required to generate a plan.

Our solution

Our current solution to this problem (which we have not yet implemented as of this writing) is to separate the execution of a token into three stages, a startup stage, a steady-state stage, and an ending stage. The startup stage performs actions to achieve the conditions that the planner intends the token to represent. The steady-state stage monitors and maintains these conditions (or signals a failure if the conditions cannot be maintained). The ending stage allows the token to perform cleanup actions before releasing control to the next token on the timeline. These stages are thus referred to as the achieve-part, the maintain-part, and the cleanup-part. Most tokens will have null actions in one or more of these three parts.

This leaves two issues to be resolved. First, how to implement the various tokens in this framework, and second, how token synchronization is actually handled. The first issue can only be handled on a token-by-token basis. The second is general across all tokens. The algorithm for executing a token in this three-phase framework is as follows:

1. Wait for the beginning of the token's start window.
2. In parallel
   (a) wait for token's pre-constraints to be true, and
   (b) check that the end of the start window has not passed. If it has, signal a failure.
3. Signal that the token has started.
4. Execute the achieve-portion of the token.
5. Spawn the maintain-portion of the token as a parallel task.
6. Wait for the start of the token's end window.
7. Wait for the token's post-conditions to be true.
8. Wait for the pre-conditions of the next token to be true, except those that refer to the end of this token.
9. Stop the maintain thread spawned in step 5, and execute the cleanup-portion of the token
10. Check that the end of the end window has not passed. If it has, signal a failure. Otherwise, signal that this token has ended.

This algorithm allows all the token types we have implemented so far to be executed within a uniform framework.

6 Examples

In the course of executing a plan, the executive needs to enforce many constraints among the components of the spacecraft. Some of these constraints are modelled by the planner and are enforced by the compatibility relationships among tokens within the plan. For example, the attitude control system (ACS) needs to maintain constant pointing during the firing of the ion propulsion system (IPS). This compatibility is expressed within the plan by a contained_by relation associated to the IPS token [1], as illustrated by the plan fragments below:

```
(PLAN-VALUE :NAME VAL-4181
 :STATE-VARIABLE (IPS IPS_SV)
 :TOKEN-TYPE  ((IPS_THRUSTING
                (IPS_TARGET_1 10)))
 :START-TIME (317 676) :END-TIME (317 676)
 :PRE-CONSTRAINTS
   (((CONTAINED_BY 0 5000000 0 5000000)
     VAL-4471 ((CONSTANT_POINTING_ON_SUN
                (IPS_TARGET_1))))...)
 :DURATION (1 5000000))
```

[1] The ACS token contains an inverse contains relation relative to the IPS token

Other essential constraints are not modelled by the planner and are known only to the executive. For example, in addition to maintaining constant pointing during IPS thrusting, the executive must change the ACS control mode to thrust vector control (TVC) once thrusting has been established. The transition between ACS control modes is modelled entirely within the executive.

The code fragment below illustrates the methods for achieving IPS thrusting at a desired level. Our code is written in ESL, the Execution Support Language (Gat 1996) being developed as the kernel for the NMRA Executive.

Note that there are multiple methods depending on the current state of execution. If the IPS is in standby mode, the ACS is commanded to change control modes only after the desired IPS thrust level has been confirmed.

```
(to-achieve (IPS-THRUSTING ips level)
 ((ips-is-in-standby-state-p ips)
   (sequence (achieve (power-on? 'ega_a))
   (command-with-confirmation
     (send_ips_set_thrust_level level))
   (command-with-confirmation
    (send_acs_change_control_mode
      :acs_tvc_mode))))
   ((ips-in-thrusting-state-p ips)
   (command-with-confirmation
    (send_ips_change_thrust_level level)))
  (t (fail :ips-achieve-thrusting)))
```

The behavior of the spacecraft and its components are never fully predictable. Because of this uncertainty, plans may contain tokens that are executed only under certain conditions. For example, a plan may specify the total amount of IPS thrusting to be achieved, with this thrusting divided among several IPS_Thrusting tokens. In some cases, the desired thrusting will be accumulated before the last IPS_Thrusting token has begun, and the executive should maintain IPS in a standby configuration rather than commanding it to thrust.

```
(defun ips-thrust-token-handler
       (ips token thrust-level)
 (with-cleanup-procedure
  (cond
   ((ips-in-standby-state-p ips) 'all-ok)
    ((ips-in-thrusting-state-p
       ips thrust-level)
       (achieve (ips-standby ips)))
    (t (sequence
         (ips-clnup/dclr-ips-unavail ips)
      (fail :ips-unavailable))))
  (with-recovery-procedures
   ((:ips-not-in-thrusting-state
     :retries *ips-retry-count*
     (reconfigure-for-token-named token)
     (retry)))
  ;; This is a a conditional token.
   (unless (db-query '(ips-thrust-achvd))
     (with-guardian
      (monitor
       (not (ips-in-thrusting-state-p
              ips thrust-level)))
       (fail :ips-not-in-thrusting-state)
      (sequence
       (achieve (ips-thrusting ips))
       (memory-wait
         '(ips-thrust-achvd))))))
 (achieve-and-maintain
   (ips-is-in-standby-state-p ips)))
```

Note that this code for the IPS-thrusting token represents the state of the DS-1 executive prior to a transition to the uniform token style described above. Nonetheless, it illustrates these principles, consisting of an achieve-part, a maintain-part, and a cleanup-part. After IPS thrusting is achieved, the monitor clause acts to maintain the IPS thrusting state. Finally, at the end of token execution, a cleanup procedure is invoked.

This code fragment illustrates other important aspects of plan execution. Execution must be robust in the face of hardware faults. For example, communication between the IPS and its remote terminal on the bus may fail temporarily during IPS thrusting. In this case, the monitor ensuring that the IPS is in thrusting state will signal a failure. Rather than failing the entire plan, the executive will attempt a reconfiguration, in this case by resetting the remote terminal.[2] If this reconfiguration succeeds, IPS thrusting can continue and the plan can be completed successfully.

In cases where the plan fails, it may be necessary to clean up after activities that are currently executing. If the plan fails during IPS thrusting, the ACS must be commanded out of TVC control mode and the IPS must be turned off. If the plan fails because the IPS has failed and cannot be successfully reconfigured, the executive declares the IPS unavailable and communicates this degraded capacity back to the planner for future planning.

7 Related Work

To the best of our knowledge, NMRA is the only integrated closed-loop system that generates and executes concurrent temporal plans in the presence of resource constraints.

[2]Our reconfigurations can draw on knowledge coded in the executive itself, or can query for assistance from an external configuration expert. In NMRA, the model-based Livingstone system (Williams and Nayak 1996) serves as such an expert.

Bresina *et al.* (1996) describe a temporal planner and executive for autonomous telescope domain. Their approach uses a single action representation whereas ours uses an abstract planning language, but their plan representation shares with ours flexibility and uncertainty about start and finish times of activities. However, their approach is currently restricted to single resource domains with no concurrency.

Currie and Tate (1991) describe the O-PLAN2 planning system, which when combined with a temporal scheduler can produce rich temporal plans. They have applied this system to a number of real-world problems including the military logistics domain. While the authors have developed an execution model for the O-PLAN plan representation (Currie and Tate 1991), the plans themselves are invoked and executed by humans, not in a closed-loop fashion. As such, their theory does not address the issue of generating current state input to the planner based on dynamic execution context.

By contrast, the Cypress system (Wilkins *et al.* 1995) and the 3T system (Bonasso *et al.* 1996) do address the closed-loop integration of planning and execution in the context of concurrency, although neither of these systems deals with temporal plans. It is interesting to compare how these systems differ from ours concerning the generation of execution context for the planner and the integration of new planning information back into execution. Cypress shares the same action formalism, called ACTS (Wilkins and Myers 1995), between planning and execution. This enables the planner to watch over execution and simulate the results forward, as discussed in section 3. The planner can detect problems in advance and send back a detailed plan refinement, and the executive can replace unexecuted portions of its current plan with new portions and continue running uninterrupted.

In 3T, the planner maintains such tight control over execution that it does not even send the full plan it has developed. Instead, it sends directives to the executive one at a time, and the executive then responds to each directive in turn. This provides an interesting solution to the problem of keeping the planner informed about execution and also to the problem of integrating new planning information into the execution context. However, this approach is problematic in our domain as it places severe time constraints on the planner so that it can decide what to do before the executive runs out of activities, and it requires the computational and informational resources to be available for planning on a continuous basis. This is a luxury we could not afford on a spacecraft, as discussed in section 3.

Other systems integrating planning and execution in real-world control systems include Guardian (Hayes-Roth 1995), SOAR (Tambe *et al.* 1995), Atlantis (Gat 1992) and TCA (Simmons 1990). These systems invoke planning as a means to answer specific questions during execution (like whether a particular treatment would take effect in time to heal the patient, which evasive maneuver will counter the opponents current attack plan, and which path to take to get to a particular room). This use of planning contrasts with our approach, in which the planner coordinates the global activity in the system. The local approach has the advantage of making use of special-purpose planners which can be built to answer narrow questions, but our global approach has the advantage of ensuring that the different activities undertaken at execution will not interact harmfully. It is not clear how the local approaches can be extended to provide similar guarantees.

8 Conclusion

A growing body of work is addressing issues of robust planning and execution in the face of failures and uncertainty. The Lockheed Underwater Vehicle (Ogasawara 1991) uses decision-theoretic planning and execution to select courses of action which maximize utility. CIRCA (Musliner *et al.* 1993) considers a set of states, actions, and critical failures to be avoided. It then inserts a set of sense-act transitions into a real-time controller to ensure that the controller will never enter the critical failure states. Cassandra (Pryor 1996a), Buridan (Draper *et al.* 1994), OPlan-2 (Currie and Tate 1991) and JIC (Drummond *et al.* 1994) all consider actions with uncertain outcomes and produce plans that enable execution-time recovery without having to take time out for replanning.

We are currently working on extending our planning approach to support such capabilities in the context of concurrent temporal plans. Our present levels of robustness are achieved using the complementary approach of flexible, abstract, and conservative plans which can be exploited by a smart executive.

A final distinction between NMRA and most other planning and execution systems is that our planner actually plans how and when it will plan for the next horizon. That is, it inserts a "plan next horizon" activity into the plan and plans other supporting activities around this goal. Such activities include information-gathering activities which will be necessary before another plan can be built. The executive then achieves these activities to enable this form of planning over multiple horizons. We believe this is a necessary capability of extended agency, and one which will become of growing concern as we design autonomous agents to achieve goals unassisted over years or decades of activity.

9 Acknowledgements

We acknowledge the contributions of other members of the DS1 Remote Agent team: Steve Chien, Mike Frank, Chuck Fry, Ed Gamble, Othar Hansson, Chris Plaunt, Sunil Mohan, Kanna Rajan, Gregg Rabideau, David Yan. Doug Bernard, Sandy Krasner, Scott Sawyer, Mike Wagner have influenced the design of the planning and execution architecture.

References

Bonasso, R. P.; Kortenkamp, D.; Miller, D.; and Slack, M. 1996. Experiences with an architecture for intelligent, reactive agents. *JETAI*. to appear.

Bresina, John; Edgington, Will; Swanson, Keith; and Drummond, Mark 1996. Operational closed-loop obesrvation scheduling and execution. In Pryor (Pryor 1996b).

Currie, K. and Tate, A. 1991. O-plan: the open planning architecture. *Artificial Intelligence* 52(1):49–86.

Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic planning with information gathering and contingent execution. In *Proceedings of AIPS94*. AAAI Press. 31–36.

Drummond, M.; Bresina, J.; and Swanson, K. 1994. Just-in-case scheduling. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Cambridge, Mass. AAAI, AAAI/MIT Press. 1098–1104.

Gat, Erann 1992. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, Cambridge, Mass. AAAI, AAAI/MIT Press.

Gat, Erann 1996. ESL: A language for supporting robust plan execution in embedded autonomous agents. In Pryor (Pryor 1996b).

Hayes-Roth, Barbara 1995. An architecture for adaptive intelligent systems. *Artificial Intelligence* 72.

Levinson, Richard 1994. A general programming language for unified planning and control. *Artificial Intelligence*. Special Issue on Planning and Scheduling.

Musliner, David; Durfee, Ed; and Shin, Kang 1993. Circa: A cooperative, intelligent, real-time control architecture. *IEEE Transactions on Systems, Man, and Cybernetics* 23(6).

Ogasawara, Gary H. 1991. A distributed, decision-theoretic control system for a mobile robot. *ACM SIGART Bulletin* 2(4):140–145.

Pell, Barney; Bernard, Douglas E.; Chien, Steve A.; Gat, Erann; Muscettola, Nicola; Nayak, P. Pandurang; Wagner, Michael D.; and Williams, Brian C. 1996. A remote agent prototype for spacecraft autonomy. In *Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation.*

Pryor, Louise 1996a. Opportunity recognition in complex environments. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland. AAAI, AAAI Press/MIT Press. 1147–1152.

Pryor, Louise, editor 1996b. *Proceedings of the AAAI Fall Symposium on Plan Execution.* AAAI Press.

Simmons, Reid 1990. An architecture for coordinating planning, sensing, and action. In *Proceedings DARPA Workshop on Innovative Approaches to Planning, Scheduling and Control.* Morgan Kaufmann: San Mateo, CA. 292–297.

Tambe, M.; Johnson, W. Lewis; Jones, R. M.; Koss, F.; Laird, J. E.; Rosenbloom, Paul S.; and Schwamb, K. 1995. Intelligent agents for interactive simulation environments. *AI Magazine* 16(1):15–39.

Wilkins, David E. and Myers, Karen L. 1995. A common knowledge representation for plan generation and reactive execution. *Journal of Logic and Computation.*

Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1995. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI* 7(1):197–227.

Williams, Brian C. and Nayak, P. Pandurang 1996. A model-based approach to reactive self-configuring systems. In *Proceedings of AAAI96.* 971–978.